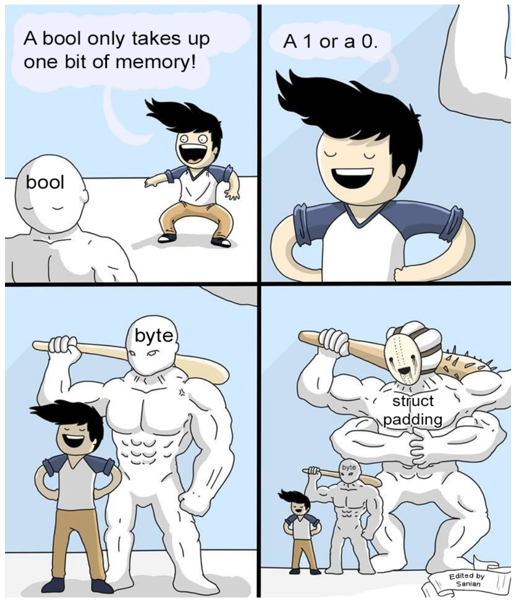
EECS 370 - Lecture 5

C to Assembly



Announcements

- HW 1
  - Posted, due next-next Monday
- P1a
  - Due Thursday
- Labs
  - Lab 2 due Wednesday
  - Lab 3 meets Fr/M

Warm-Up Excercise

- Write ARM assembly code for the following C expression:
  - (assume an int is 4 bytes and that struct elements are stored contiguously)

```
          4                     1      1
struct { int a; unsigned char b, c; } y;
y.a   = y.b + y.c;
```

- Assume that a pointer to y is in X1.

```
LDURB  X2 , [X1, #4]      STURW  X4 , [X1 , #0]
LDURB  X3 , [X1, #5]
add    X4 , X2, X3.
```

Warm-Up Excercise

- Write ARM assembly code for the following C expression:
  - (assume an int is 4 bytes and that struct elements are stored contiguously)

4 | X1 : y

```
struct { int a; unsigned char b, c; } y;
y.a  = y.b + y.c;
X2     X3    X4
```

- Assume that a pointer to y is in X1.

```
LDURB X3 [X1,#4]      SDURW X2 [X1, #0]
LDURB X4 [X1,#5]
add   X2 X3 X4
```

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: ISA - storage types, binary and addressing modes
- Lecture 3 : LC2K
- Lecture 4 : ARM
- **Lecture 5 : Converting C to assembly – basic blocks**
- Lecture 6 : Converting C to assembly – functions
- Lecture 7 : Translation software; libraries, memory layout

Agenda

- **Memory alignment**
    - Aligning Structs
- Control flow instructions
    - C-code examples
- Extra Problems

Calculating Load/Store Addresses for Variables

Datatype	size (bytes)
char	1
short	2
int	4
double	8

```
short   a[100];
char    b;
int     c;
double  d;
short   e;
struct  {
   char f;
   int  g[1];
   char h;
} i;
```

- *Problem*: Assume data memory starts at address 100, calculate the total amount of memory needed

a = 2 bytes * 100 = 200

b = 1 byte

c = 4 bytes

d = 8 bytes

e = 2 bytes

i = 1 + 4 + 1 = 6 bytes

total = 221, right or wrong?

Memory layout of variables

- Compilers don't like variables placed in memory arbitrarily
- As we'll see later in the course, memory is divided into fixed sized **chunks**
  - When we load from a particular chunk, we really read the whole chunk
  - Usually an integer number of words (32 bits)
- If we read a single char (1 byte), it doesn't matter where it's placed

0x1000	0x1001	0x1002	0x1003
'a'	'b'	'c'	'd'

```
ldurb [x0, 0x1002]
```

- Reads [0x1000-0x1003], then throws away all but 0x1002, fine

Memory layout of variables

- BUT, if we read a 32-bit integer word, and that word starts at 0x1002:

0x1000	0x1001	0x1002	0x1003
0xFF	0xFF	0x12	0x34

0x1004	0x1005	0x1006	0x1007
0x56	0x78	0xFF	0xFF

- First we need to read [0x1000-0x1003], throw away 0x1000 and 0x1001, **then** read [0x1004-0x1007]
- Need to read from memory twice! Slow! Complicated! **Bad!**

Solution: Memory Alignment

Poll: Where can chars start?

- Most modern ISAs require that data be aligned
  - An N-byte variable must start at an address A, such that (A%N) == 0
- For example, starting address of a 32 bit **int** must be divisible by 4

int can start here

int CAN'T start here

0x1000	0x1001	0x1002	0x1003
0x12	0x34	0x56	0x78

- Starting address of a 16 bit **short** must be divisible by 2

short can start here

int CAN'T start here

0x1000	0x1001	0x1002	0x1003
0x12	0x34	0x56	0x78

Golden Rule of Alignment

You can't break into smaller size

- Every (primitive) object starts at an address divisible by its size
- "Padding" is placed in between objects if needed

```
char  c;
short s;
int   i;
```

0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
[c]	[padding]	[s]		[i]			

- But what about non-primitive data types?
  - Arrays? Treat as independent objects
  - Structs? Trickier…

Agenda

- Memory alignment
  - **Aligning Structs**
- Control flow instructions
  - C-code examples
- Extra Problems

Problem with Structs

- If we align each element of a struct according to the Golden Rule, we can still run into issues
  - E.g.: An array of structs

```
char c; 1000

struct {
  char c; 1001
  int i; 1004-1007
} s[2]; 1001-1007
```

Amount of padding is different across different instances

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	100A	100B	100C	100D	100E	100F
c	s[0].c	[pad]	[pad]	s[0].i				s[1].c	[pad]	[pad]	[pad]	s[1].i			

- Why is this bad?
- It makes "for" loops very difficult to write!
  - Offsets need to be different on each iteration

Structure Alignment

- Solution: in addition to laying out each field according to Golden Rule…
  - Identify largest (primitive) field
    - Starting address of overall struct is aligned based on the largest field
    - Padded in the back so total size is a multiple of the largest primitive

```
char c;

struct {
  char c;
  int i;
} s[2];
```

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	100A	100B	100C	100D	100E	100F
c	[pad]	[pad]	[pad]	s[0].c	[pad]	[pad]	[pad]	s[0].i				s[1.c]	[pad]	[pad]	[pad]

Guaranteed to lay out each instance identically

Structure Example

1000

1000
1004-1007
1008-1011
1012-1015
1016
1018-1019

```
struct {
   char w;
   int x[3];
   char y;
   short z;
}
```

1
4
= int x_0, int x_1,
int x_3
2
1000-1019 (20)

B

<u>Poll:</u> What boundary should this struct be aligned to?

a) 1 byte
b) 4 bytes
c) 12 bytes
d) 2 bytes
e) 19 bytes

- Assume struct starts at location 1000,
  - char w → 1000
  - x[0] → 1004-1007, x[1] → 1008 – 1011, x[2] → 1012 – 1015
  - char y → 1016
  - short z → 1018 – 1019

Total size = 20 bytes!

Datatype	size (bytes)
char	1
short	2
int	4
double	8

Calculating Load/Store Addresses for Variables

```
short a[100];
char b;
int c;
double d;
short e;
struct {
  char f;
  int g[1];
  char h;
} i;
```

[100 299]
[300 300]
[304 307]
[312 319]
[320 321]
[324 325]
[328 331]
[332 332]
[324 335]

We need to add padding to ensure that the total size of this struct is divisible by four.

- *Problem*: Assume data memory starts at address 100, calculate the total amount of memory needed

a = 200 bytes (100-299)

b = 1 byte (300-300)

c = 4 bytes (304-307)

d = 8 bytes (312-319)

e = 2 bytes (320-321)

struct: largest field is 4 bytes, start at 324

f = 1 byte (324-324) g: 324 -327

g = 4 bytes (328-331) f: 328 - 328

h = 1 byte (332-332) h: 329 - 329

i = 12 bytes (324-335) i: [324 - 331]

236 bytes total!! (compared to 221, originally)

6+2=8+1=9 12-9=3

Data Layout – Why?

- Does gcc (or another compiler) reorder variables in memory to avoid padding?
- No, C99 forbids this
  - Memory is laid out in order of declaration for structs
- The programmer (i.e., you) are expected to manage data layout of variables for your program and structs.
- Two optimal strategies:
  - Order fields in struct by datatype size, smallest first
  - Or by largest first

Agenda

- Memory alignment
  - Aligning Structs
- **Control flow instructions**
  - C-code examples
- Extra Problems

ARM/LEGv8 Sequencing Instructions

- Sequencing instructions change the flow of instructions that are executed
  - This is achieved by modifying the program counter (PC)
- Unconditional branches are the most straightforward
  they ALWAYS change the PC and thus “jump” to another instruction out of the usual sequence
- Conditional branches

If (condition_test) goto target_address

condition_test examines the four flags from the processor status word (SPSR)
target_address is a 19 bit signed <u>word</u> displacement on current PC

LEGv8 Conditional Instructions

- Two varieties of conditional branches
  1. One type compares a register to see if it is equal to zero.
  2. Another type checks the condition codes set in the status register.

PC + offset : In LEGv8, we don't need PC+1 + offset.

Conditional branch	compare and branch on equal 0	CBZ X1, 25	if (X1 == 0) go to PC + 100 (25×4)	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	CBNZ X1, 25	if (X1 != 0) go to PC + 100	Not equal 0 test; PC-relative branch
	branch conditionally	B.cond 25	if (condition true) go to PC + 100	Test condition codes; if true, branch

- Let's look at the first type: CBZ and CBNZ
  - CBZ: Conditional Branch if Zero
  - CBNZ: Conditional Branch if Not Zero

LEGv8 Conditional Instructions

- CBZ/CBNZ: test a register against zero and branch to a PC relative address
  - The relative address is a 19 bit signed integer—the number of instructions. Recall instructions are 32 bits of 4 bytes

We can branch "forward" or "backward"

Conditional branch	compare and branch on equal 0	CBZ X1, 25	if (X1 == 0) go to PC + 100	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	CBNZ X1, 25	if (X1 != 0) go to PC + 100	Not equal 0 test; PC-relative branch
	branch conditionally	B.cond 25	if (condition true) go to PC + 100	Test condition codes; if true, branch

- Example: CBNZ X3, Again
  - If X3 doesn't equal 0, then branch to label "Again"
  - "Again" is an offset from the PC of the current instruction (CBNZ)
  - Why does "25" in the above table result in PC + 100?

LEGv8 Conditional Instructions

- Example: What would the offset or displacement be if there were two instructions between ADDI and CBNZ?

```
0   Again:   ADDI    X3, X3, #-1
1                    --------------
2                    --------------
3                    CBNZ    X3, Again
```

$\boxed{-12}$

PC = pc + offset

3 − 0 = 3

offset = 3×4 = 12

PC' = 0   PC = 3.

offset = 3

~~B~~

Poll: What's the offset?

a) -16

b) -12

c) -4

d) -3

e) 0

We will go back 12 bytes,

But when we code the offset,

We only care how many construction should

I go forward or backward.

LEGv8 Conditional Instructions

- Motivation:
  - Some types of branches makes sense to check if a certain value is zero or not
    - while(a)
  - But not all:
    - if(a > b)
    - if(a == b)
  - Using an extra **program status register** to check for various conditions allows for a greater breadth of branching behavior

LEGv8 Conditional Instructions Using FLAGS

0 or 1 one bit

- FLAGS: NZVC record the results of (arithmetic) operations Negative, Zero, oVerflow, Carry—not present in LC2K
- We explicitly set them using the "set" modification to ADD/SUB etc.
- Example: ADDS causes the 4 flag bits to be set according as the outcome is negative, zero, overflows, or generates a carry

Category	InstructionExample		Meaning	Comments
Arithmetic	add	ADD X1, X2, X3	X1 = X2 + X3	Three register operands
	subtract	SUB X1, X2, X3	X1 = X2 – X3	Three register operands
	add immediate	ADDI X1, X2, 20	X1 = X2 + 20	Used to add constants
	subtract immediate	SUBI X1, X2, 20	X1 = X2 – 20	Used to subtract constants
	add and set flags	ADDS X1, X2, X3	X1 = X2 + X3	Add, set condition codes
	subtract and set flags	SUBS X1, X2, X3	X1 = X2 – X3	Subtract, set condition codes
	add immediate and set flags	ADDIS X1, X2, 20	X1 = X2 + 20	Add constant, set condition codes
	subtract immediate and set flags	SUBIS X1, X2, 20	X1 = X2 – 20	Subtract constant, set condition codes

ARM Condition Codes Determine Direction of Branch

- In LEGv8 only ADDS / SUBS / ADDIS / SUBIS / CMP /CMPI set the condition codes FLAGs or condition codes in PSR—the program status register
- Four primary condition codes evaluated:
  - N – set if the result is negative (i.e., bit 63 is non-zero)
  - Z – set if the result is zero (i.e., all 64 bits are zero)
  - ~~C – set if last addition/subtraction had a carry/borrow out of bit 63~~
  - ~~V – set if the last addition/subtraction produced an overflow (e.g., two negative numbers added together produce a positive result)~~
- Don’t worry about the C and V for this class

ARM Condition Codes Determine Direction of Branch--continued

Encoding	Name (& alias)	Meaning (integer)	Flags
0000	EQ	Equal	Z==1
0001	NE	Not equal	Z==0
0010	HS (CS)	Unsigned higher or same (Carry set)	C==1
0011	LO (CC)	Unsigned lower (Carry clear)	C==0
0100	MI	Minus (negative)	N==1
0101	PL	Plus (positive or zero)	N==0
0110	VS	Overflow set	V==1
0111	VC	Overflow clear	V==0
1000	HI	Unsigned higher	C==1 && Z==0
1001	LS	Unsigned lower or same	!(C==1 && Z==0)
1010	GE	Signed greater than or equal	N==V
1011	LT	Signed less than	N!=V
1100	GT	Signed greater than	Z==0 && N==V
1101	LE	Signed less than or equal	!(Z==0 && N==V)
1110	AL	Always	Any
1111	NV[†]	Always	Any

Need to know the 7 with the red arrows

```
CMP   X1, X2
B.LE Label1
```

For this example, we branch if X1 is >= to X2

Conditional Branches: How to use

pseudo instruction

- CMP instruction lets you compare two registers.
  - Could also use SUBS etc. and then throw away the result (eg: to X_{31} zero registers)
    - That could save you an instruction. But then it will update the status register flags
- B.cond lets you branch based on that comparison.

- Example:

```
CMP   X1, X2
B.GT  Label1
```

check condition

if $X_1 > X_2$, then branch to Label 1.

We don't have to specify any register, we can still tell the previous comparison $X_1 > X_2$, and then branch based of that.

The advantage: (split this across multiple instructions). since we don't specify any other registers, So we can make the label pretty long. So can branch far away

  - Branches to Label1 if X1 is greater than X2.

Agenda

- Memory alignment
  - Aligning Structs
- Control flow instructions
  - **C-code examples**
- Extra Problems

Branch—Example

- Convert the following C code into LEGv8 assembly (assume x is in X1, y in X2):

if condition is true, then go to the nextline, if not jump to the other position.
However, Pc in default is go to the next line.

```
int x, y;
if (x == y)
  x++;
else
  y++;
// …
```

invert

```
      cmp   X1  X2
      b.ne  else
      addi  X1, X1, #1
      b     end
  or  b.eq  end
else  addi  X2  X2  #1
end
```

unconditional branch.
no matter what, go to end label.

Branch—Example

- Convert the following C code into LEGv8 assembly (assume x is in X1, y in X2):

x1 x2

```
int x, y;
if (x == y)
  x++;
else
  y++;
// …
```

CMP X1 X2

B.NE else

if ADDI X1 X1 #1

else ADDI X2 X2 #1

Branch—Example

- Convert the following C code into LEGv8 assembly (assume x is in X1, y in X2):

```c
int x, y;
if (x == y)
  x++;
else
  y++;
// …
```

Using Labels

```
     CMP  X1, X2
     B.NE L1
     ADD  X1, X1, #1
     B    L2
L1:  ADD  X2, X2, #1
L2:  …
```

Note that conditions in assembly are often the inverse of the "if" condition. Why?

Without Labels

```
CMP     X1, X2
B.NE    3
ADD     X1, X1, #1
B       2
ADD     X2, X2, #1
```

Assemblers must deal with labels and assign displacements

Loop—Example

// assume all variables are long long integers (64 bits or 8 bytes)
// i is in X1, start of a is at address 100, sum is in X2

```
sum = 0;
for (i=0 ; i < 10 ; i++) {
  if (a[i] >= 0) {
    sum += a[i];
  }
}
```

```
          MOV     X1, XZR    } initialize i and sum = 0
          MOV     X2, XZR
Loop1:    CMPI    X1, #10
          B.EQ    endLoop
          LSL     X6, X1, #3
          LDUR    X5, [X6, #100]
          CMPI    X5, #0
          B.LT    endif
          ADD     X2, X2, X5
endif:    ADDI    X1, X1, #1
          B       Loop1
endLoop:
```

# of branch instructions
= 3*10 + 1= 31

a.k.a. while-do template

Loop—Example

// assume all variables are long long integers (64 bits or 8 bytes)
// i is in X1, start of a is at address 100, sum is in X2

```
sum = 0;
for (i=0 ; i < 10 ; i++) {
  if (a[i] >= 0) {
    sum += a[i];
  }
}
```

x1 x2 x4 x3 address

MOV X1 XZR
MOV X2 XZR
for CMPI X2 #10
B.GE endfor
LSL X3 X1 #3
LDUR X4 X3 #100
CMPI X4 #0
B.LT endif
ADD X1 X1 X4
endif ADDI X2 X2 #1
B. for.
endfor

# of branch instructions
= 3*10 + 1= 31

a.k.a. while-do template

Agenda

- Memory alignment
  - Aligning Structs
- Control flow instructions
  - C-code examples
- **Extra Problems**

Extra Example: Do-while Loop

// assume all variables are long long integers (64 bits or 8 bytes)
// i is in X1, start of a is at address 100, sum is in X2

```
sum = 0;
for (i=0 ; i < 10 ; i++) {
  if (a[i] >= 0) {
    sum += a[i];
  }
}
```

```
          MOV     X1, XZR
          MOV     X2, XZR
Loop1:    LSL     X6, X1, #3
          LDUR    X5, [X6, #100]
          CMPI    X5, #0
          B.LT    endif
          ADD     X2, X2, X5
endIf:    ADDI    X1, X1, #1
          CMPI    X1, #10
          B.LT    Loop1
endLoop:
```

# of branch instructions
= 2*10 = 20

a.k.a. do-while template

Extra Example: Do-while Loop

// assume all variables are long long integers (64 bits or 8 bytes)
// i is in X1, start of a is at address 100, sum is in X2

```
sum = 0;
for (i=0 ; i < 10 ; i++) {
  if (a[i] >= 0) {
    sum += a[i];
  }
}
```

# of branch instructions
= 2*10 = 20

a.k.a. do-while template

Extra Problem – For Your Reference

- Write the ARM assembly code to implement the following C code:

```
// assume ptr is in X1
// struct {int val; struct node *next;} node;
// struct node *ptr;

if ((ptr != NULL) && (ptr->val > 0))
  ptr->val++;
```

Extra Problem

- Write the ARM assembly code to implement the following C code:

```
// assume ptr is in X1
// struct {int val; struct node *next;} node;
// struct node *ptr;

if ((ptr != NULL) && (ptr->val > 0))
  ptr->val++;
```

```
    cmp r1, #0
    beq Endif
    ldursw  r2, [r1, #0]
    cmp r2, #0
    b.le Endif
    add r2, r2, #1
    str r2, [r1, #0]
Endif : ….
```

Extra Class Problem

- How much memory is required for the following data, assuming that the data starts at address 200 and is a 32 bit address space?

```
int a;
struct {double b, char c, int d} e;
char* f;
short g[20];
```

<u>Poll:</u> How much memory?

a) x < 40 bytes
b) 40 < x < 50 bytes
c) 50 < x < 60 bytes
d) 60 < x bytes

Next Time

- More C-to-Assembly
  - Function calls
- Lingering questions / feedback? Post it to Slido